Report

이름: 김문겸

학번: 20220124

Lab: 1

Homework1-1: bitNor

```
int bitNor(int x, int y) {
        //to be implemented
        return (~x & ~y);
}
```

NOR 을 AND 와 NOT 으로 구현하는 문제이다. 이를 위해 De Morgan's Laws 을 적용하였다.

Homework1-2: isZero

```
int isZero(int x) {
        //to be implemented
        return !x;
}
```

!은 오직 0 인 경우와 0 이 아닌 경우를 판단하기 때문에 사용하였다. x 가 0 일 경우 !0 = 1 이 되고, 0 이 아닐 경우 0 을 return 한다.

Homework1-3: addOK

```
int addOK(int x, int y) {
        //to be implemented
        int check = (x + y) >> 31;
        int result = ~(((x >> 31) ^ (y >> 31)) ^ (((x >> 31) ^ check) | ((y >> 31) ^
check)));
        return 1 & result;
}
```

overflow 의 여부를 확인하는 함수이다. 이 함수를 구현하기 위해서는 x, y 그리고 x+y 의 msb, 부호를 확인해보아야 한다고 판단했다. overflow 가 발생하는 경우는 다음과 같다. 양수+양수=음수, 음수+음수=양수인 경우이다. 이 경우를 제외한 양수+음수 또는 음수+양수의 경우 overflow 를 발생시키지 않는다. 따라서 이 문제에서 확인해주어야 하는 것은 'x 와 y 의 부호가 같은가?'와, 'x, y 의 부호가 같다면, x+y 의 부호와 x, y 의 부호가 같은가?'를 확인해주어야 하는 것이다. 따라서 각각 (x >> 31) ^ (y >> 31)와 ((x >> 31) ^ check) | ((y >> 31) ^ check))로 확인을 해주었다.

만약 x, y 의 부호가 다르다면, 즉 overflow 가 발생하지 않는 상황이라면, (x >> 31) ^ (y >> 31) 위치에서 해당 항은 1111...11 이 되고, ((x >> 31) ^ check) | ((y >> 31) ^ check)) 위치에서는 필연적으로 x, y 중 하나는 x+y 와 부호가 같을 것이므로 해당 항 또한 1111...11 이 될 것이다. 그렇게 되면 (((x >> 31) ^ (y >> 31)) ^ (((x >> 31) ^ check) | ((y >> 31) ^ check)))은 0000...00 이 되고, result 는 그 값을 NOT 연산하기 때문에 result = 1111.11 이 된다. 해당 값을 1 과 AND 연산하게 되면 값은 1 이 나오게 된다.

만약 부호가 같다면, (x >> 31) ^ (y >> 31)값은 0000...00 이 나오고, (((x >> 31) ^ check) | ((y >> 31) ^ check)))의 경우 오버플로우가 발생한다면 1111...11 이 나오게 될 것이다. 그렇게 되면 result 값은 0000...00 이 나오고, 해당 값을 1 과 AND 연산하면 0 이 나오게 된다. 만약 오버플로우가 발생하지 않는다면 x, y, x+y 부호가 같은 것이므로 (((x >> 31) ^ check) | ((y >> 31) ^ check)))의 값은 0000...00 이 나오고, result = 1111...11 이 나온다. 이 값을 1 과 AND 연산하면 1 이 나오게 되어, 모든 경우에서 oveflow 의 발생 유무를 확인할 수 있게 된다.

Homework1-4: absVal

```
int absVal(int x) {
        //to be implemented
        return ((x + (x >> 31)) ^ (x >> 31));
}
```

절댓값의 경우 음수를 양수로 바꾸어 return 해야 한다. 하지만 양수는 그대로 return 해야 한다. 따라서 양수와 음수를 나타내는 비트 표현에서 절댓값이 같은 두 음수와 양수를 비교했을 때, 양수 표현이 음수의 NOT 표현보다 1 크다는 것을 이용하였다. 거 x>>31 은 양수의 경우 0 의 값을 가지지만, 음수의 경우 -1 의 값을 가진다. 따라서 x+(x>>31)은 음수의 경우 x-1, 양수의 경우 x 가 되게 된다. 이제 x>>31 과 NOR 연산하게 되면, 음수의 경우 1111...11 과 NOR 연산하게 되어 (~x)+1 이 된다. 이는 음수 x 의 절댓값과 같다. 양수의 경우 0000...00 과 NOR 연산을 진행하므로 x 값 그대로 나오게 된다.

Homework1-5: logicalShift

```
int logicalShift(int x, int n) {
        //to be implemented
        return (x >> n) & ~(((1 << 31) >> n) << 1);
}
```

일반적으로 shift 연산을 하게 되면 arithmetic shift 가 적용된다. 따라서 이 문제에서는 arithmetic shift 를 한 후, logical shift 상 0 으로 바뀌어야 하는 왼쪽 자리의 1 들을 AND 연산을 통해 0 으로 바꿔주도록 구현하였다. 1<<31 을 하게 되면 MSB 자리에 1 이 위치한 bit 가 만들어지고, 이를 n 만큼 arithmetic shift 하게 되면 기존 x 의 MSB 가 arithmetic shift 로 인해 변경되었던 왼쪽 부분의 1 들을 만들어내게 된다. 여기서 다시 왼쪽으로 1 만큼 shift 하게 되면, x 의 비트들을 제외한, arithmetic 으로 인해 만들어진 1 들을 구현하게 된다. 이것을 NOT 연산하게 되면, 해당 부분은 모두 0 이 되고, 이를 (x >> n)의 1 들과 AND 연산하게 되면 0 이 된다. 따라서 Logical shift 를 구현할 수 있게 된다.